Introduction to Object-oriented Programming

Patrick D. O'Brien and Daniel C. Halbert

Digital Equipment Corporation
Object-Based Systems Group
HLO 2-3/M08
77 Reed Road
Hudson, MA 01749 USA
(617) 568-5627

June, 1988

The following are trademarks of Digital Equipment Corporation: Trellis, Ultrix, VAX, VAXmate.

1 Introduction

The software industry is under enormous pressure to develop more complex information systems and to reduce the time and effort required to develop these systems. Object-oriented programming is an emerging technology that addresses these problems of complexity and productivity. It provides abstraction mechanisms that help deal with the complexities of large software systems, and it increases programmer productivity by enhancing software maintainability, extensibility, and reusability.

Object-oriented programming is becoming a key buzzword of the late 1980's, reminiscent of the way "structured programming" was a hot topic in the 1970's. It is quickly gaining popularity and is already in widespread use. Most software experts hold out more hope for object-oriented programming than for any of the other current technical fads. It is finding its greatest acceptance in application areas where the pressure to deal with complexity and productivity problems is most acute.

2 Core concepts of object-oriented programming

Many of the ideas behind object-oriented programming have roots in the Simula 67 programming language. Object-oriented programming also borrows heavily from abstract-data-type programming languages which were developed in the 1970's, such as CLU and Alphard. Smalltalk was the first interactive, display-based implementation of an object-oriented system. Although there is some agreement on the fundamental principles of object-oriented programming, the programming language community is still actively experimenting with many of the concepts.

Object-oriented programming combines data and behavior in a single entity called an *object*. The data in an object consists of fields that reference other objects. The operations associated with an object collectively characterize its behavior. Objects with the same behavior are grouped into *types* (also called *classes*) and are known as *instances* of their types. A type characterizes the behavior of its instances by describing the operations that can manipulate those objects.

For example, consider a type called `Bank_account`. A `Bank_account` object (that is, an instance of type `Bank_account`) would describe a particular bank account and contain fields that store the current balance, the account number, and other such information. The `Bank_account` type would implement operations such as `deposit`, `withdraw`, and so on.

This notion of a type is derived from abstract-data-type programming languages. A type defines an abstraction. When you create an object, you create a black box. You can find out about and change the state of the object only through the operations its provides to you. You see an interface, but you cannot see the implementation.

Object-oriented programming languages extend this idea of abstract data types by adding *subtyping* and *inheritance*. Subtyping allows one type to be a subtype of another, with the implication that if `B` is a subtype of `A`, an object of type `B` may be used wherever an object of type `A` may be used. In other words, objects of type `B` are also of type `A`. Some languages permit a subtype to have multiple parent types (*supertypes*). The relationships between the types define a *type hierarchy*, which can be drawn as a graph. In the simple example shown in Figure 1, both `Checking_account` and `Savings_account` are subtypes of `Bank_account`.

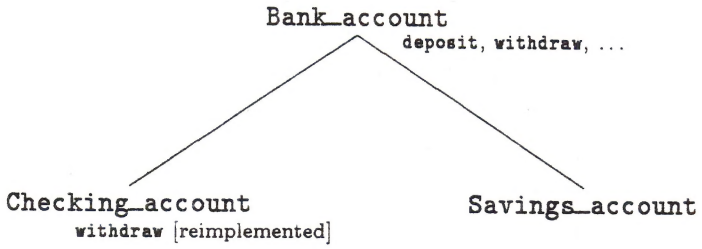

Figure 1: A simple type hierarchy.

Intimately connected with subtyping is the concept of *inheritance*. A subtype may share or *inherit* various characteristics of its supertype(s). The subtype in turn may pass on its own or its inherited characteristics to its subtypes. The characteristics that can be inherited usually include the fields defined in the supertype and the operations provided by it. For example, in the hierarchy given above, `Bank_account` defines storage for an account balance in its instances, and also defines operations for manipulating the balance. These definitions would be inherited by `Checking_account` and `Savings_account`.

Inheritance is very powerful. It lets you factor out common characteristics from a number of related types and implement those characteristics in one place, the supertype. With inheritance you usually get the ability to *override* an inherited operation to change what it does or how it does it. For instance, in Figure 1, the `withdraw` operation is reimplemented. It might add the ability to provide overdraft protection, or it might just implement withdrawals in a different way.

Notice there are two `withdraw` operations. We say that the `withdraw` operation name is *overloaded*, because it stands for more than one operation. The types of the arguments determine which operation to call. Sometimes the type is not known until run-time, and so the language must be able to do *late binding*, meaning that it must be able to choose which operation to call at run-time.

Automatic storage management, though not essential, is so useful that it is frequently provided by object-oriented languages. When you create an object, you allocate some

storage. When you are finished using an object, you would like that storage to be recycled. But since an object usually points to other objects, which may be shared, it is frequently difficult to know when an object is no longer needed. If the system provides a way of reclaiming storage automatically (say, by using garbage collection), then you will no longer have to worry about keeping track of what storage you can reclaim.

3 Creating types and type hierarchies

When you design and implement a program, you map application concepts into programming language concepts. In conventional procedural programming, this usually means looking at functions the program should perform and mapping these functions into program code. Thus, programs end up being modularized and decomposed based on procedural criteria. In object-oriented programming, however, you start by concentrating on the data the program should manipulate. Your program design consists of types and the operations you perform on objects of those types.

How do you choose what should be a type? In many cases, the entities in the application domain map naturally into types. These types may correspond to physical objects like airplanes and employees, or they may correspond to concepts needed by the application, like fuel consumption records and salary histories. In other cases, coding the application may introduce internal concepts that do not correspond to real-world entities, such as various internal tables. As long as clear abstractions can be constructed for these concepts, there is no reason they cannot be types as well.

When you define a type, you should practice good software engineering principles, as you would with any other style of programming. Types should be modular, interdependencies between types should be minimized, and implementation details should not be exposed to the user of the type. You should also think about designing types that are general-purpose, so that you can reuse existing types in future programs.

Often the set of types needed by an application turns out to include a number of types that are related to each other. Type hierarchies help you model these relations. The key idea of subtyping is that a program can be constructed by first modeling, in terms of types, the most general concepts in the application domain, and then dealing with special cases through more specialized subtypes.

Subtyping is most frequently used for modeling conceptual hierarchies. Sometimes the hierarchy produced is a model of the real world. For instance, `Jet` and `Turboprop` might be subtypes of `Airplane`. `Employee` might be a subtype of `Person`.

Another common use for subtyping is to guarantee that types present a certain interface. You can define a type that specifies certain operations. Many of the operations may be unimplemented, with only their interfaces described. You would not actually create any

instances of this type, but instead you would use it when creating subtypes. For instance, `Input_device` might be the common supertype for `Mouse`, `Tablet`, and `Keyboard` (see Figure 2). Each kind of input device would implement some operations and add new ones. As another example, you might define a type, `Dictionary`, and then implement this type in two ways, say, `Hash_table` and `Binary_search_table`. The subtyping in this case is being used for implementation only: the two subtypes behave identically if viewed as black boxes.

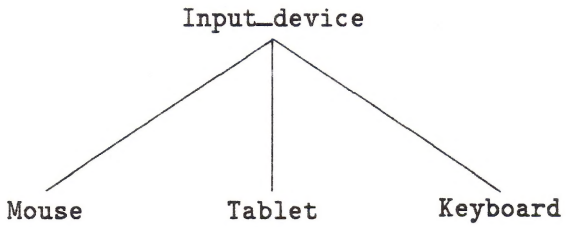

Figure 2: Input_device hierarchy.

There are other ways to use subtyping for structuring code, but these are the most common (see [7] for a more detailed discussion on the use of types and inheritance in object-oriented programming).

4 Why is object-oriented programming useful?

Object-oriented technology addresses two of the most pressing problems in the software industry: programmer productivity and our inability to build large complex systems that are reliable. The complexity arises from trying to model complex real world systems. This task involves designing and implementing a system by mapping application concepts into programming language concepts. The resulting programs exhibit both complex data structures and complex algorithms. For large systems, there is also substantial complexity in trying to manage the integration of numerous separate components. Thus a comprehensive solution must deal with all of these complexities.

The productivity problem is not a question of building programs more quickly. The real issue, which is closely related to the complexity problem, is how to enhance software maintainability, extensibility, and reusability. Enhancing maintainability involves reducing the time and effort it takes for you to fix malfunctioning software. Improving extensibility allows you to evolve a program, adding new features over time and having the program grow gracefully. Increasing the reusability of software results in your not having to "reinvent the wheel" every time a new program is developed.

The object-oriented methodology enables you to do program design at a high level. Since you can use types to model application entities, the structures in object-oriented programs can closely resemble the structures in the application domain. The modular decomposition of the program will be based on the types of objects the program manipulates, not on the functions the program performs. In conventional control-based programming, you are forced to modularize programs based on procedural criteria. This often leads to program structures that are radically different from the structures in the application domain.

Object-oriented programming also lets you take a "building-block" approach to system development. You can build general-purpose types that can be reused, giving you additional leverage when adding new functionality. In addition, you can use inheritance to factor out common behavior from a number of similar types. If you have designed the type hierarchy with generality in mind, you will provide for easy extensibility, because you will be able to add new types that build on existing ones.

The modularity provided by types in object-oriented languages also encourages encapsulation, allowing you to hide implementation details. You can separate the specification and implementation of a type. This enhances maintainability and extensibility by encouraging a well-designed type structure, which minimizes and localizes the interdependencies among types, and greatly reduces the integration costs of building a complex system.

5 Object-oriented databases

The class of applications requiring database system support is quickly extending beyond the more traditional data processing applications into areas such as computer-aided design (CAD and CASE), office automation, and artificial intelligence. These applications have complex data structuring needs, significantly different data accessing patterns, and special performance requirements. Conventional programming methodologies are not necessarily appropriate for these applications, and similarly, conventional data management systems may not be appropriate for managing their data.

Conventional data management systems perform very well for many kinds of applications. They are good at batch processing large amounts of homogeneous data, like monthly credit-card billings. They are also good for high-transaction-rate applications such as ATM networks. Recently, relational database systems have been developed to provide good support for ad hoc queries where the user declares *what* to retrieve from the database as opposed to *how*. But the processing of data in applications such as CAD or CASE does not fit any of these categories. The design data in a mechanical or electrical CAD database is not homogeneous: it consists of complex relationships between many types of data. The transactions in a CASE system can take hours, or days, not just seconds. Moreover, database access for these applications is dominated by graph walking as

opposed to *ad hoc* querying.

The object-oriented programming methodology has proven to be particularly well suited for applications having complex data structuring needs. But object-oriented languages are usually memory-based and typically support a single user or application. Objects either do not persist between program executions or are saved in a primitive manner that makes it difficult to support sharing or reliability (for example, copying a program's object memory to disk). Object-oriented databases rectify this situation by combining the benefits of object-oriented programming with the data management facilities of database management systems.

An object-oriented database combines the major object-oriented programming concepts of data abstraction, encapsulation, and type hierarchies, and the database concepts of storage management, sharing, reliability, consistency, and associative retrieval. Designing an object-oriented database system that satisfies the performance requirements of these applications and that merges well with the object-oriented paradigm is not an easy task. But recent work in integrating object-oriented concepts with database technology is beginning to show a lot of potential.

6 Looking for more information?

If this article has piqued your interest in object-oriented programming, you may want to follow up by reading some other articles or books. In this section we suggest a few articles and provide pointers to other sources of information, though this is not intended to be a comprehensive bibliography.

To fully appreciate the benefits that object-oriented programming provides, you have to understand the problems that the software industry is facing. Fred Brooks' article, "No Silver Bullet" [2], is a very readable account of why software engineering is such a difficult task. He also talks about some of the new approaches, such as object-oriented programming, and comments on the promise that they hold for improving the plight of software engineering. Another article along these lines has appeared in Business Week [3] and is particularly suited for readers who are not software experts.

After understanding the problems of software engineering, you are ready to be immersed in the gospel of object-oriented programming. For a light sermon and a glimpse at some of the activity in the area try John Verity's article "The OOPS Revolution" [21]. Brad Cox is a little more emphatic and more detailed in his article "Message/Object Programming: An Evolutionary Change in Programming Technology" [4].

For good technical introductions to object-oriented programming we can suggest two articles. Mark Stefik and Dan Bobrow contrast several languages including Lisp-based object-oriented extensions in their article "Object-Oriented Programming: Themes and

Variations" [1]. Bjarne Stroustrup uses examples from C++ when answering the question "What is object-oriented programming?" [19].

Data abstraction is arguably the most important concept of object-oriented programming. Mary Shaw's "Abstraction Techniques in Modern Programming Languages" [15] provides a succinct technical overview of data abstraction. Inheritance is perhaps the second most important concept of object-oriented programming. See Alan Snyder's "Encapsulation and Inheritance in Object-Oriented Programming Languages [16] for a good discussion of inheritance.

These last two papers focus on the importance of data abstraction and inheritance and how different languages provide features to support these concepts. We show how to put these concepts into practice in our article "Using types and inheritance in object-oriented programming" [7].

Object-oriented database research is becoming an increasingly important area. Carlo Zaniolo et al. provide a brief introduction in "Object-oriented database systems and knowledge systems" [23]. Jacob Stein also introduces some of the major concepts in his paper "Object-oriented programming and databases" [17]. For an overview of how object-oriented databases differ from relational databases and why some application areas are looking at object-oriented databases to meet their data management needs as opposed to using relational databases, see [9].

If you get past this point and still want more information, there are still plenty of sources. The next place to look would be in the proceedings of the last two Object-Oriented Programming Systems, Languages, and Applications (OOPSLA) conferences [11] [12]. There is also a European equivalent of OOPSLA called ECOOP [5]. There is a new journal devoted entirely to object-oriented programming [22] and the first issue has a very good article that contrasts several of the more important object-oriented languages. For object-oriented databases try the ACM Transaction on Office Information Systems special issue on object-oriented systems [20] or the proceedings of the First International Workshop on Object-Oriented Database Systems [8]. Last but not least, IEEE has put together two volumes of papers on object-oriented programming from a variety of sources [13].

7 How to get started?

Should you start experimenting with an object-oriented system? Do these systems have the capabilities and performance that your applications demand? Deciding if and when to jump on the bandwagon is difficult and there are tradeoffs to be made. Object-oriented programming languages and databases are emerging technologies. As the technologies mature, they will begin to catch existing systems in terms of performance and breadth of capabilities.

Other promising technologies of the past have failed to yield their expected benefits—how will you know if object-oriented systems are going to deliver on the promises of improved productivity and reliability? You will never know if you do not evaluate them. There are several object-oriented systems available on Digital hardware that you can begin experimenting with today. Our advice is to get going!

When deciding which system to use there are several factors to keep in mind. First, consider the requirements of the applications you are developing. Integration with existing systems, database needs, and performance should be taken into account. Second, evaluate what it is you are getting with a particular object-oriented system. An object-oriented system often includes a lot more than just a programming language and a compiler. Important components include a programming environment, an extensive library of general-purpose types, and a database manager. Finally, be wary — many systems say they are object-oriented, but are just trying to hitch a ride on the buzzword. They can lack key concepts, such as inheritance. You can simulate object-oriented programming in many languages, but it may be very awkward and error-prone.

Perhaps the best known object-oriented system is Smalltalk. Its strengths include a pure object-oriented model (everything is an object), a powerful programming environment, a good type library, lots of applications, and implementations on several platforms. Its weaknesses include an unconventional syntax, lack of type checking (which is essential for maintainability), very little integration with other systems, and performance (it is interpreted as opposed to compiled). We do not know of an efficient implementation of Smalltalk on a VAX although there is one that will run on the VAXmate.

The closest system to Smalltalk in terms of a powerful prototyping environment is Digital's Trellis system (see [14] and[10]). The Trellis/Owl language improves upon some of Smalltalk's shortcomings by providing strong typing, multiple inheritance, and a more conventional syntax. Trellis has an incremental compiler which provides the quick turnaround of an interpreter with the performance of a compiler. Its programming environment is similar to Smalltalk's but provides additional capabilities and takes advantage of the information that a strongly-typed language offers. Trellis runs on both VMS and Ultrix and it provides capabilities such as a call-out facility to integrate with existing applications. Please contact us for information about Trellis/Owl.

There are several object-oriented extensions to Lisp that you can experiment with. The Lisp community seems to be standardizing on CLOS, which is an object-oriented extension to Common Lisp. CLOS seems to be a well thought out design and it should be available soon. At this time there are no plans for a standard programming environment for CLOS.

The one language that appears to be gaining the most widespread support is C++ (see [18]). Its major strength (and limitation) is that it tries to be as upwardly compatible as possible with C. C++ is a strongly-typed language like Trellis/Owl. It is evolving at this time but will eventually include important language features that Trellis/Owl provides,

such as multiple-inheritance, parameterized types, and exception handlers. C++ is just a language, however; there is no programming environment or standard type library. Current implementations are mainly preprocessors that generate C code, making debugging very awkward. Some more recent implementations are developing compilers, but we are not aware of any incremental compiler efforts. Licenses for Ultrix and VMS versions of C++ are available from a couple of companies.

Another alternative is to use languages such as ADA or Modula-2. These languages provide support for abstract data types and encapsulation but do not have type hierarchies. The absence of a type hierarchy is an important deficiency but you can still adhere to an object-oriented style using simple coding conventions,

Whatever choice you make, or even if you continue to use you current programming language and environment, we believe that you should make the effort to understand object-oriented programming. It may take considerable effort on your part, but the benefits will be well worth the expense.

References

[1] Mark Stefik and Daniel G. Bobrow. "Object-oriented programming: themes and variations." *AI Magazine*, Vol. 6, No. 4, Winter 1986.

[2] Frederick P. Brooks, Jr. "No Silver Bullet—Essence and accidents of software engineering." *Computer*, Vol. 20, No. 4, April 1987.

[3] "The Software Trap: Automate—or else." *Business Week*, May 9th, 1988.

[4] Brad J. Cox. "Message/Object Programming: An Evolutionary Change in Programming Technology." *IEEE Software*, January 1984.

[5] *Proceedings of the European Conference on Object-Oriented Programming* (ECOOP), Paris, France, June 1987.

[6] A. Goldberg and D. Robson. *Smalltalk-80: The language and its implementation.* Reading, Massachusetts: Addison-Wesley, 1983.

[7] Daniel C. Halbert, and Patrick D. O'Brien. "Using Types and Inheritance in Object-oriented Programming." *IEEE Software*, September 1987.

[8] *International Workshop on Object-Oriented Database Systems*, Pacific Grove, California, September 1986.

[9] Patrick D. O'Brien. *Coors Justification*, Digital Equipment Corporation internal technical report, April 1988.

[10] Patrick D. O'Brien, Daniel C. Halbert, and Michael F. Kilian. "The Trellis Programming Environment." *OOPSLA-87 Proceedings*, Orlando, Florida, October 1987.

[11] *OOPSLA '86 Conference Proceedings*, Portland, Oregon, September 1986.

[12] *OOPSLA '87 Conference Proceedings*, Orlando, Florida, October 1987.

[13] G. Peterson Ed. *IEEE Tutorial on Object-oriented Computing Concepts.* Volume 1 ISBN-08186-0821-8, Volume 2 ISBN-08186-0821-6.

[14] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An Introduction to Trellis/Owl. *OOPSLA86 Proceedings*, Portland, Oregon, September 1986.

[15] Mary Shaw. "Abstraction Techniques in Modern Programming Languages," *IEEE Software*, October 1984.

[16] Alan Snyder. "Encapsulation and Inheritance in Object-Oriented Programming Languages." *OOPSLA86 Proceedings*, Portland, Oregon, September 1986.

[17] Jacob Stein. "Object-oriented programming and databases." *Dr. Dobb's Journal*, March 1988.

[18] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 1986.

[19] Bjarne Stroustrup. "What is object-oriented programming?" *IEEE Software*, May 1988.

[20] *ACM Transactions on Office Information Systems*, Vol. 5, No. 1, January, 1987.

[21] John W. Verity. "The OOPS Revolution." *Datamation*, May 1st, 1987.

[22] Richard S. Weiner, ed. *Journal of Object-Oriented Programming*, SIGS Publications, New York, NY.

[23] Carlo Zaniolo, et al. *Object-oriented Database Systems and Knowledge Systems*. MCC technical report DB-038-85, July 1985.